International Hellenic University, Serres Campus
Robotics and Intelligent Systems Lab

The Robin SoC processor, MCU Edition: basic specifications

The Robin open project

Prof. Ioannis Kalomiros
19/8/2019
Document version 0.1

The Robin SoC processor, MCU Edition: basic specifications

The Robin SoC processor, MCU Edition: basic specifications

1. Overview

1.1 General overview

Robin System on Chip is an educational soft processor, designed using Hardware Description Language (VHDL). The Robin project is developed by the Robotics and Intelligent Systems Lab of the International Hellenic University, Serres Campus, Greece.

The **v0** version (*microcontroller edition*) of the processor is designed as a special purpose 8-bit microcontroller. It is a RISC processor and it follows some of the principles of MIPS processors. Also, it follows some of the principles of the Instruction Set Architecture implemented in Microchip PIC microcontrollers.

Being an educational processor, the Robin System-on-Chip in its basic version implements a Very Reduced Instruction Set Computer (V-RISC), with basic "move" and "alu" instructions, and basic branch and conditional branch ability. It features an 8-level stack for nested procedure calls.

The project involves undergraduate and post graduate students who work on the hardware and the software parts. Beside the processor core, various interfaces and peripherals will be developed, such as parallel I/O ports, timers, PWM modules, serial port, I2C port etc. In the future, the system will incorporate on chip an ADC and flash memory. The hardware will be customized to configure both Intel and Xilinx FPGAs. As a first step, the processor will be implemented using a low cost MAX10 device by Intel.

Software design will include a simple assembler and a programmer. As a future project, a Python or C compiler is considered.

As an open project, the Robin project invites external contributors who apply in order to develop specific tasks. For this purpose, a special web site will be created.

Robin processor documentation will be part of the teaching material in lessons on digital system design and computer architecture, in IHU program studies. Since it constitutes an introductory and easy to understand, yet powerful processor, teaching Robin will be assessed in terms of its virtues as an educational paradigm.

1.2 Architectural overview

The main processor core consists of the instruction decoder, a register file, an ALU and a control unit. The general design of the processor datapath is shown in Fig. 1. The main data bus, busA, is input to the register file, while busB and busC is output from the register file and input to the **ALU**. Immediate addressing is provided through a 2:1 multiplexer which can choose an 8-bit literal value as an input argument to the ALU unit.

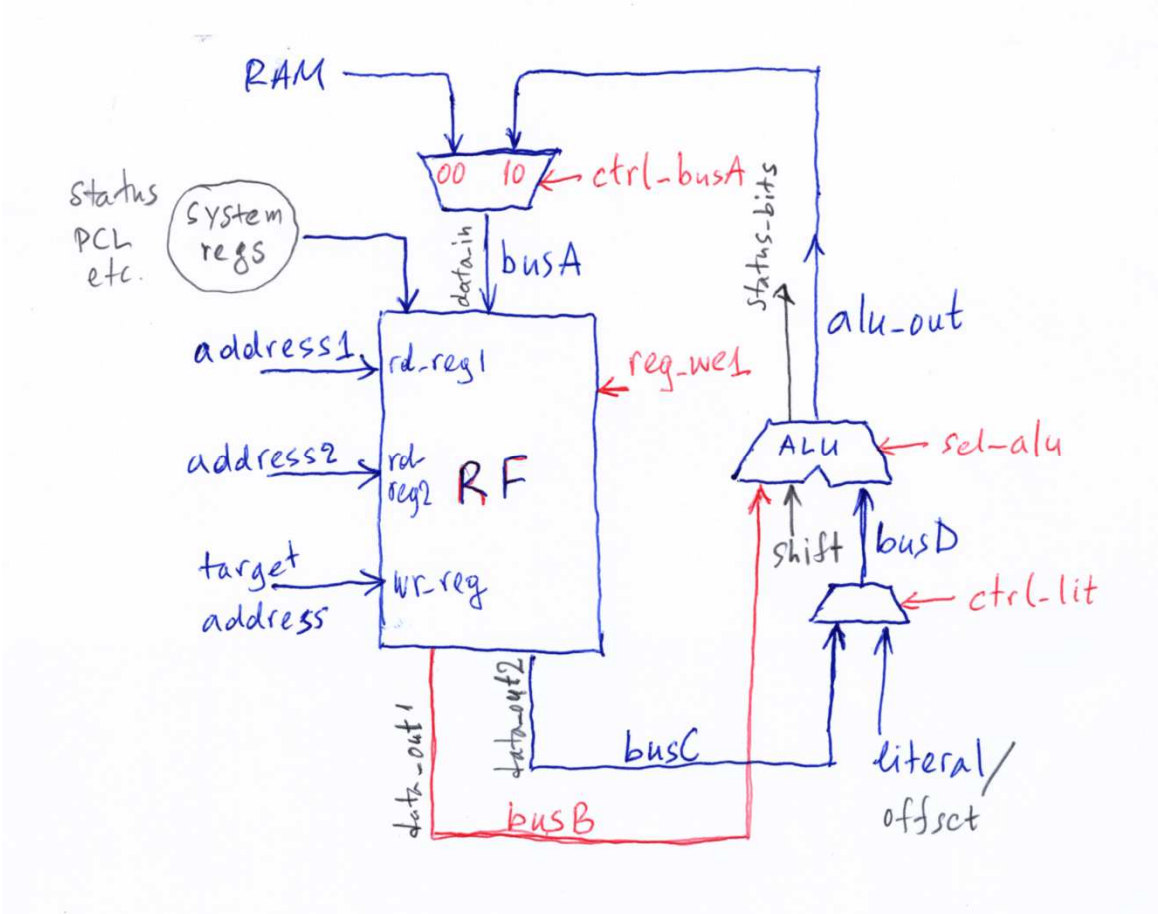


Fig. 1 Main datapath of the Robin processor core

The **register file** consists of 32 8-bit registers divided into subsets of system registers, general registers and peripheral registers. It is designed as a multi-port RAM, with two address ports (address1 and address2) for reading and one address port for writing (target address). In this way, it reads simultaneously two registers (reg1, reg2) from the register file and outputs their contents onto busB and busC. It can write the data bus (busA) into the target address, which is either address1 or W reg (see below).

The decoder and control units are shown in Fig. 2. The **decoder unit** receives the instruction word from the program memory and decodes the opcode and operants of the command. At each instruction cycle a new instruction is fed into the decoder circuit and it is segmented into appropriate fields (opcode, register or ram addresses, literal value and a function bit). The opcode and the function bit (fb) are input to the main **control unit FSM**, which decodes the control signals at each execution step, depending on the opcode and fb. The main control signals are mux-select and alu-select signals, register write-enable signal and instruction memory-enable signal. In the extended processor version with RAM, a RAM rd_en/w_en signal is also decoded.

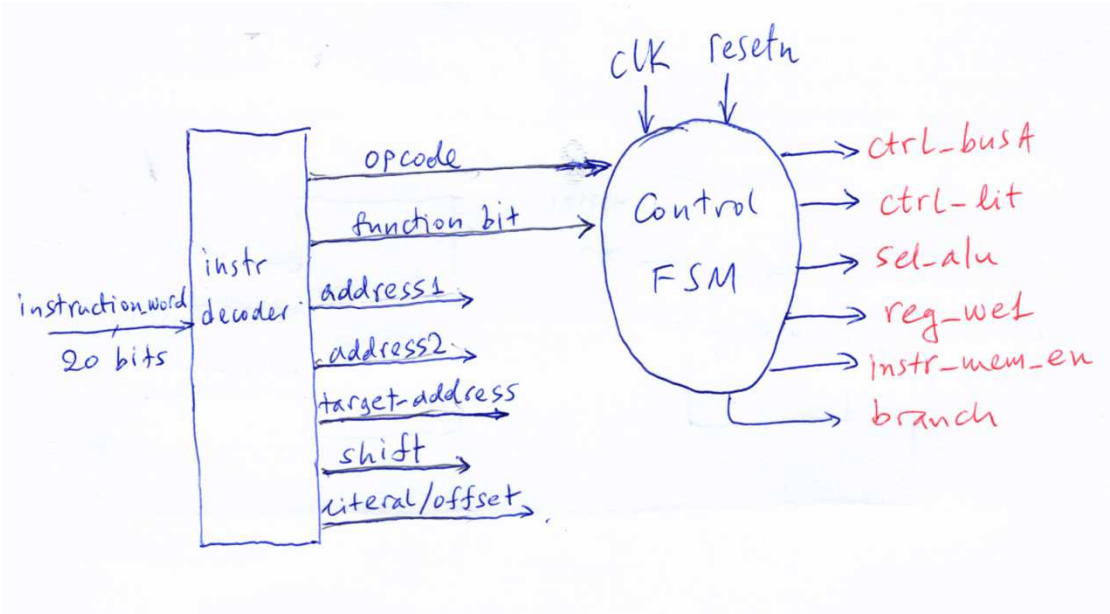


Fig. 2 The decoder unit and the controller FSM.

The control unit is designed as a four-cycle Finite State Machine following the basic principle that all instructions are executed in the same four periods of the main clock. The four phases are the typical fetch-decode-execute-store states, each one lasting one clock period.

The Robin processor core with its main blocks is shown in the diagram of Fig. 3. It consists of the decoder unit, the control FSM, the register file and the ALU unit. In the same diagram, the program counter circuit is also shown.

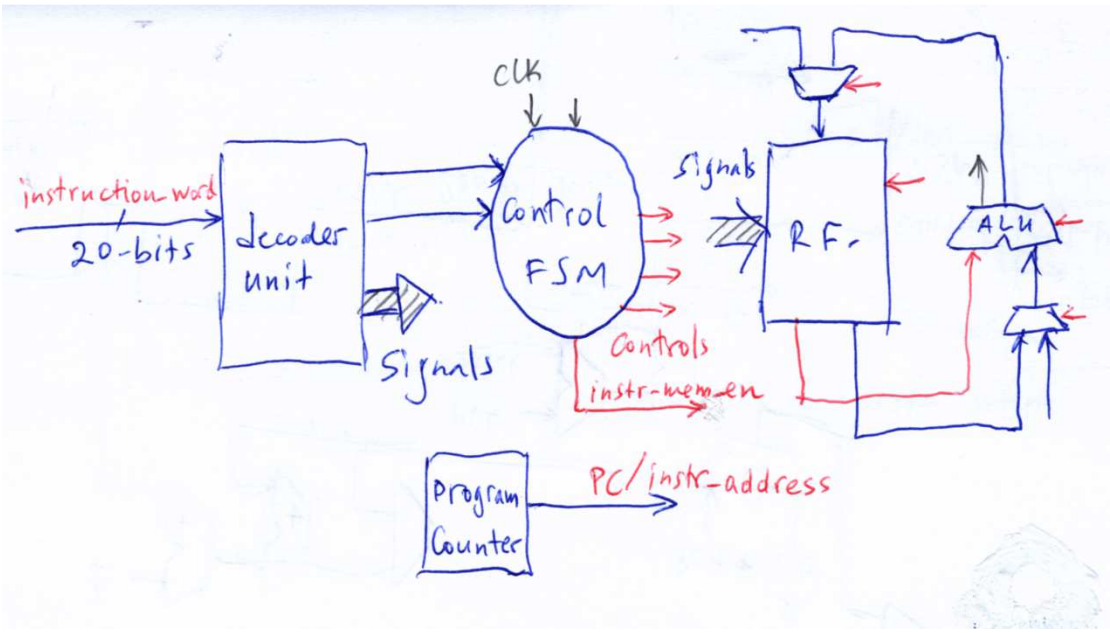


Fig. 3 The main processor core as a total of four main units. The program counter is implemented as a separate circuit.

The next instruction address is decoded in the decode phase of the execution of the current command. For this purpose a program counter (pc) is incremented at the fetch period of the current command. The program counter value is used to derive the instruction address which is input to the instruction memory block. The instruction memory-enable control signal is set in the store phase of the current command, initiating the reading of the next command. The program counter circuit implements additional logic for the execution of branch and jump commands. This logic will be discussed later.

An idle state initiates the control fsm where all signals are assigned their initial values. The system falls in the idle state when program execution is completed with an end command.

In its microcontroller edition (v0), the processor follows a strict Harvard architecture, with a separate Instruction Memory. The basic instruction memory has a capacity of 1024 instruction words, each word having a width of 20 bits. Therefore, instruction address is 10 bits. The fields of the instruction word are explained below. In the expanded processor version, maximum program length can be up to 8K. Utilizing a second program memory bank, it can go up to 16K (see the role of function bit, footnote 3, in section 4, jump operation).

In Fig. 4, the processor is presented as a top level entity with both the instruction memory block and the processor core block. The core consists of the units shown in Fig. 3.

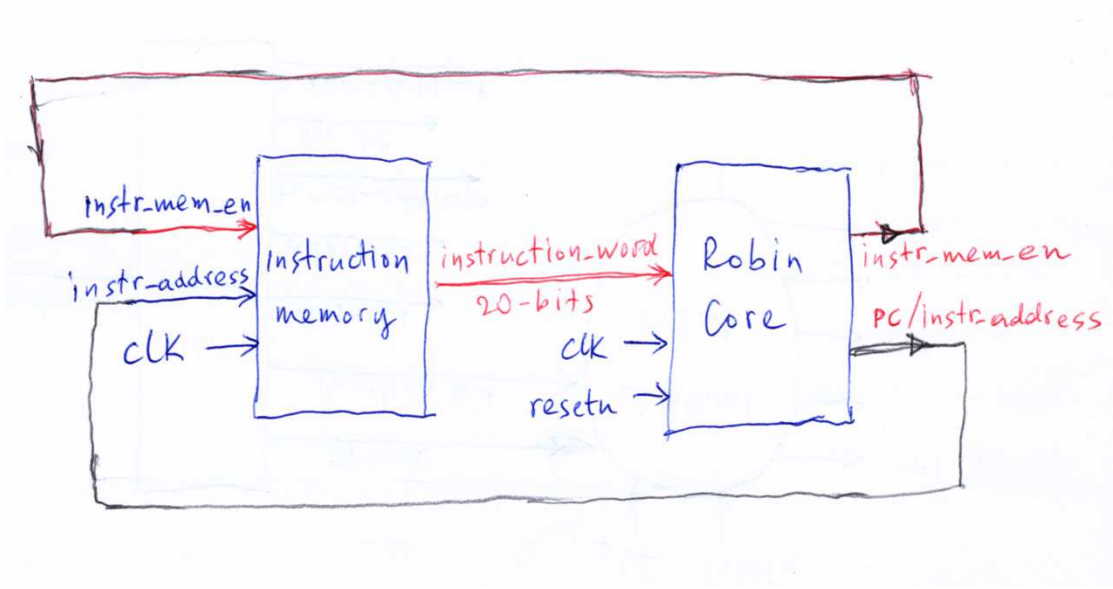


Fig. 4 Top level entity of the processor, with instruction memory and core circuits.

1.4 Versions of the Robin SoC - MCU Edition

In the basic version of the processor, the main processor core consists of an instance of the decoder, the ALU, the register file and the control FSM. It also implements the control multiplexers and the program counter with jump ability and conditional branch. A top-level entity instantiates the instruction memory and the main core.

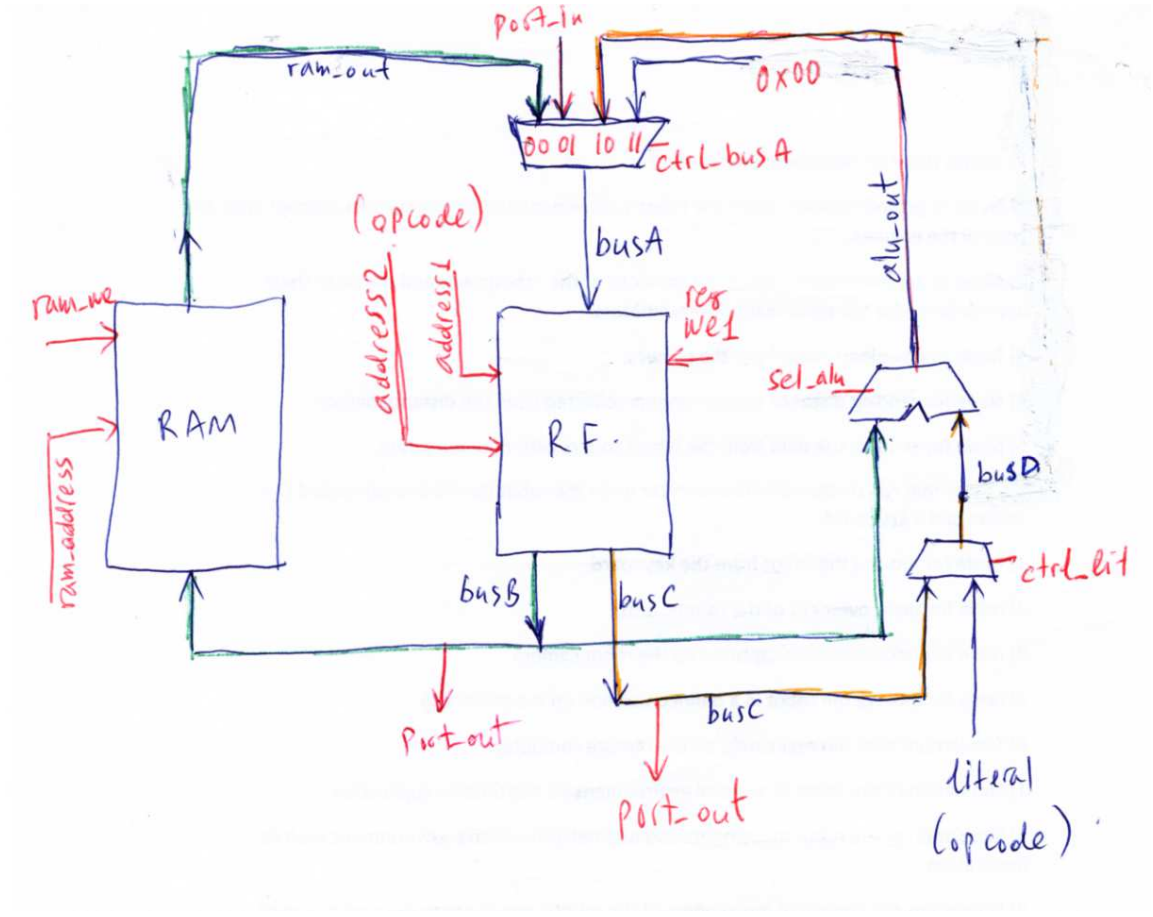


Fig. 5 Expanded processor with RAM

In the expanded processor version, a RAM memory block is added and offers storing capabilities (Fig. 5). The capacity of the processor memory is 256 bytes in the most basic version. In this version, reg2 in the instruction keeps the memory address to be read or to be written. In base+offset address configuration the memory capacity can be up to 512 bytes and in the base & offset version (see operation of ld, st commands and the possible uses of the base address register) the memory capacity can reach 64K.

2. Instruction word

The binary instruction word is consisted of the fields shown ton Fig. 6. Different types of commands use different fields. The main scheme, however, begins with an opcode, followed by the5-bit address of the first register and ends with a 5-bit address of register 2, according to the form:

opcode <address1>, <address2>,

where address1 is placed into address field1 and address2 into address field 2. Opcode is always the first 6-bit field of the binary command.

The form of different instructions with their opcodes is shown in Section 4, Table II.

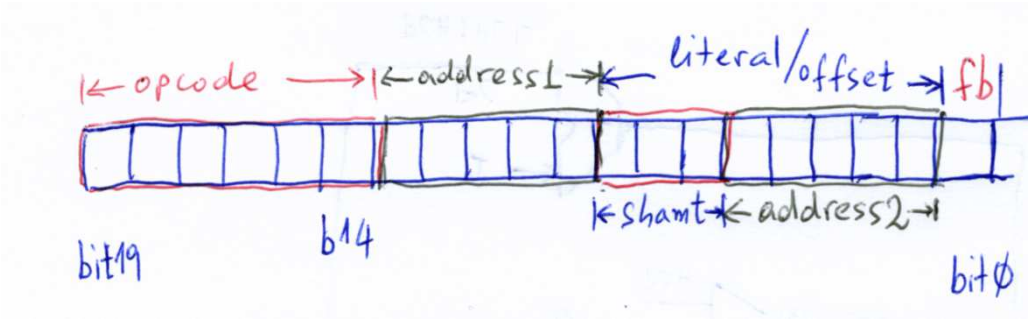


Fig. 6 Fields of instruction word

A 3-bit field between the two address fields is called "shamt" and is mainly used to host the shift parameter in shift instructions. It is also combined with the address2 field, in order to represent an 8-bit literal value. A literal or immediate value represents a number instead of an address. In this sense, the command

movi <address1>, literal (e.g. movi 0x0A, 0x28)

moves the literal value 0x28 into register with address 0x0A. In this case the assembler puts the address 0x0A into address field1 and the literal value 0x28 into the 8-bit combination shamt & address field2. The operator & means concatenation.

The literal value in the shamt & address field 2 can host an 8-bit offset for a branch operation. Details are given in Section 5.

The instruction: shift <address1>, <address2>, shift_parameter

shifts the content of register with address2 by the shift parameter and places the result into address1. the assembler places the shift parameter into shamt field. In this sense, the shift parameter can be in the range 0 to 7.

The final bit in the binary instruction word is the function bit or fb. This bit defines one among alternative ways a command is executed. Details of the alternatives offerd by the function bit are given in Table II (see footnotes).

A total of 64 commands can be implemented by the above binary instruction scheme. The register file can only be 32 registers, because of the 5 bits in address fields.

Finally, address field 1 in combination with shamt field and address field 2 can host an absolute address (maximum 13 bits) for a jump operation.

As a conclusion, different types of commands make different use of the instruction fields. Depending on the meaning of the various fields, we recognize three basic types of instructions:

1) opcode-reg-reg, 2) opcode-reg-literal 3) opcode-address

3. Register file

3.1 Presentation of the register file

The register file (RF) consists of 32 resgisters accessible by 5-bit fields in the binary instruction word. These fields are positioned after the opcode. The registers are divided in the following groups, according to their role:

TABLE I. THE BASIC REGISTER MAP

address	Register name	Description	comments
00000	W reg	working register	system register
00001	Status reg	zero, carry, gt,lt,eq	system register
00010	IC	Interrupt Flags	system register
00011	PCL	Program counter low	system register
00100	PCH	Program counter high	system register
00101	Base address reg	keeps RAM base address	system register
00110	reg1	general register	general user register
00111	reg2	"	"
01000	reg3	"	"
01001	reg4	"	"
01010	reg5	"	"
01011	reg6	"	"
01100	reg7	"	"
01101	reg8	"	"
01110	reg9	"	"
01111	reg10	"	"
10000	reg11	"	"
10001	reg12	"	"
10010	reg13	"	"
10011	reg14	"	"
10100	reg15	"	"
10101	reg16	"	"
10110	preg1	Can be assigned to a peripheral	peripheral register
10111	preg2	"	"
11000	preg3	"	"
11001	preg4	"	"
11010	preg5	"	"
11011	preg6	"	"
11100	preg7	"	"
11101	preg8	"	"
11110	preg9	"	"
11111	preg10	"	"

system registers (w, status, interrupt flags, program counter low, base address, reserved). They are six registers.

general registers (general (source/target) registers reg1-reg16, used as either sources or targets. They are sixteen registers.

peripheral registers (for expansion peripherals. They may be grouped in two or four registers per peripheral). They are ten registers. They can be used as normal general registers if they are not assigned to user peripherals.

3.2 Description of system registers

Among the **system registers**, the following play important role:

W register can be used as a general accumulator to transfer alu results among registers. Its use is combined with function bit of the instruction word (see section 4, footnotes). If the function bit (fb) is set to 1, then the alu result will be stored in the W register. If fb=1, then the result will be stored in register reg1 (field 1 in the instruction word).

Base address register can be used to keep a starting address of an array in memory and is combined with an offset in reg2 of ld or st command. In one version, it can hold a starting address and add the offset (base + offset). In another version, it can hold the 8 MSBs of an address and concatenate them with the offset in reg2 of an ld or st command (extends RAM up to 64K).

STATUS register is a system register that stores the basic flags derived from unsigned arithmetic during alu operation. STATUS bits are detailed below:

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
LS	shift	gt	eq	lt	OV	Zero (Z)	Carry (C)

Carry flag (C) is set when a carry/borrow bit is produced into the N-bit of an unsigned addition or subtraction. **Zero** flag (Z) is set when the alu result is zero. **Overflow** (OV) bit denotes overflow when it occurs and should be taken into account in signed arithmetic.

Comparison bits **gt**, **eq**, **lt** are only set after a sub or cmp operation. In fact "cmp reg1, reg2" performs a subtraction reg1-reg2 and sets the comparison bits based on the carry and zero bit result, namely gt = carry XOR zero, eq = zero, lt = NOT(carry) (see also entity alu_unit.vhd in Appendix A).

Shift bit (bit5) hosts the shifted MSB of a user register after the execution of a shift-left (sl) command. It hosts the shifted LSB of a user register after the execution of a shift-right (sr) command.

Literal/offset **sign bit** (**LS**) hosts the most significant bit of a literal/offset value. Depending on its value and the value of the produced carry bit upon the execution of a branch command, the pc_high field is incremented, decremented or remains unaltered.

STATUS register remains unaltered until a new alu operation affects its bits.

4. The Instruction Set

The basic instruction set is presented in Table II.

TABLE II. THE V-RISC INSTRUCTION SET

Category	Instruction name	Assembly code	opcode	Funbit	example	Operation	Comments
Data transfer (register-register)	move immediate	movi	000001	0	movi reg1, 0x28	w/reg1 <- literal	Move literal value to reg1 (immediate)
	move	mov	000010	0	mov reg1, reg2	w/reg1 <- reg2	Move content of reg2 to reg1 (register direct)
	move indirect	mind	000000	0	mind reg1, *base	w/reg1 <- [reg2]	Move the content of register address in base reg into reg1 (register indirect)
Data transfer (register-memory)	load	ld	000011	0/1[1]	ld reg1, *reg2	reg1<-[reg2]	Load reg1 with the content of address in reg2 (memory indirect - ram version)
	store	st	000100	0/1[1]	st reg1, &reg2	reg1->[reg2]	Store reg1 to the address in reg2 - ram version)
Arithmetic	add	add	000101	0/1[2]	add reg1, reg2	w/reg1 <- reg1+reg2	Affects zero, ov and carry bits
	sub	sub	000110	0/1[2]	sub reg1, reg2	w/reg1<-reg1-reg2	Affects gt/eq/lt, zero, ov and carry bits
	add with carry c	addc	000111			w/reg1 <- reg1 + reg2 + carry	Affects zero, ov and carry bits
	subtract with borrow	subc	001000			w/reg1 <- reg1 - reg2 - carry	Affects gt/eq/lt, zero, ov and carry bits
	add immediate	addi	001001	0/1[2]	addi reg1, 0x28	w/reg1<-reg2+literal	
	sub immediate	subi	001010	0/1[2]	sub reg1, 0x28	w/reg1<-reg1-literal	
Logical	and	and	001011	0/1[2]	and reg1, reg2	w/reg1<- reg1 and reg2	Bitwise and
	or	or	001100	0/1[2]	or reg1, reg2	w/reg1<- reg1 or reg2	Bitwise or
	and immediate	andi	001101	0/1[2]	andi reg1, 0x28	w/reg1 <- reg1 and literal	
	or immediate	ori	001110	0/1[2]	ori reg1, 0x28	w/reg1 <- reg1 or literal	
	Shift left	sl	001111	0	sl reg1, reg2, 2	w/reg1<-reg2 << shift (0-7)	shifted MSB in shift-bit b5 of status reg, i.e. status(5)<-reg2(7)
	Shift right	sr	010000	0	sr reg1, reg2, 1	w/reg1<-reg2 >> shift(0-7)	Shifted MSB in b5 of status reg. i.e. status(5)<-reg2(0)

	compare	cmp	010001	1	cmp reg1, reg2	Set gt/lt/eq bits of STATUS reg (W <- reg1-reg2)	Compare register contents and set status bits. Affects W.
Conditional and unconditional branch	Branch on equal	beq	010010	1	beq offset	If *reg1=*reg2 in previous cmp command, then pc<-pc+offset	Branch type 1: PC-relative, +/- 127 commands
	Branch on greater than	bgt	010011	1	bgt offset	If *reg1>*reg2 in previous comp then pc<-pc+offset	Branch type 1: PC-relative, +/- 127 commands
	jump	jmp	010100	0/1[3]	jmp &address	jump to address	Branch type 2: absolute 10-bit program address
	Call subroutine	call	010101	0	call Label	Push PC in stack and jump to subroutine	Absolute addressing 10-bit
	Return from subroutine	ret	010110	0	ret	Pop PC from stack and return from subroutine	
other	Clear register	clr	010111	0	clr reg	Clear all bits of a register	Only in expanded set
	No operation	nop	011000	0	nop	Do nothing	Only in expanded set
	End execution	end	111111	0	end	Go to the idle state and sleep	

[1] With Function bit =1, use base-address register as base address and content of reg1 as offset. Memory is extended to 64K in base & offset version. It is extended to 512 bytes in base + offset version. With Function bit = 0, only 256 bytes of memory are accessible.

[2] With Function bit = 1 the result is stored in working register w.

[3] With Function bit = 0 the branch address is in the first page of program memory (0-8191). With Function bit = 1 the branch address is in the second page of program memory (8192-16383).

5. Implementation of branch and jump commands

5.1 Branch, type 1

One version of branch command implements small jumps with an offset +/- 127, within the current and the next program memory segment. The first program memory segment is 0-255, the second is 256-511, the third is 512-767 and the last is 768-1023. These segments hold for the basic processor version, with 1K program memory (1Kx20bits). For larger program memories, segments continue likewise, every 256 words.

A branch on equal command, for example, is

beq <offset>,

where offset is an 8-bit literal. This command follows a cmp command:

cmp <reg1>, <reg2>

which sets the comparison bits of the STATUS register, based on the magnitude relation of the contents of registers reg1 and reg2.

If the condition reg1=reg2 holds, then the branch command actually makes use of the add operation of the alu and adds the 8-bit offset with the contents of the pc_low register. The pc_high field is also incremented or decremented automatically or remains constant, depending on the offset sign (LS) and the carry out (C) of the add operation. In this way, the program counter points to its next value, within the current or the next or the previous segment. Such a relatively small jump up or down 127 commands is usually enough for small programs.

The command bgt <reg1>, <reg2> implements exactly the same operation, on the condition that [reg1] > [reg2].

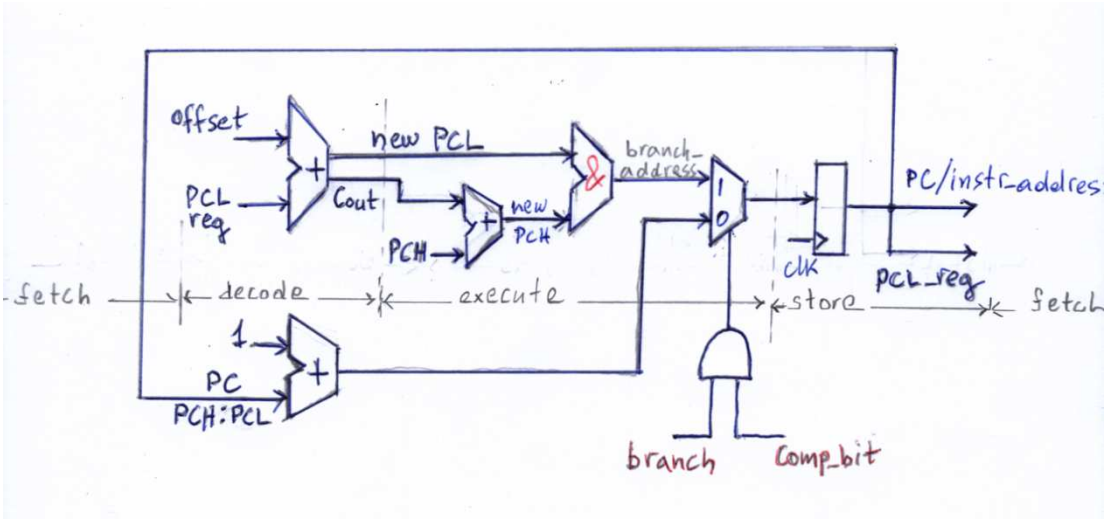


Fig. 7 Implementation of the program counter logic. Without branch, the PC is incremented at each command. With branch, an offset is added to the PC_Low value and the PC_high is adjusted accordingly.

A version of the program counter circuit that implements the branch commands is shown in Fig. 7.
The pc_high field (1:0) is changed during the execute phase, according to the logic described below. The pc_high change depends on the value of the carry and LS bits in status register, after the addition of the PC_LOW register value and the offset value in the ALU (see Fig. 7)

carry	LS	pc_high field change	Add to pc_high
1	0	+1	01
1	1	0	00
0	1	-1	11
0	0	0	00

Based on the table above, the logic used to derive the addition parameter h_bits in order to produce an additive change to pc_high is described below:

```
with (carry XOR LS) select

h_bits <= ls & '1' when '1',

          "00"    when '0';
```

The full program counter circuit is described in VHDL in entity robin_core, in the Appendix.

5.2 Branch, type 2

The second version of branch commands is "jump" or "jmp" command. It receives as operand a 10-bit absolute address and sets the program counter to this address. The 10-bit operand consists of the 2 lower bits of the field1 operand & the "shamt" operand & the 5-bit field2 operand. In this way we can reach a maximum absolute address of 13 bits, if all the bits of field 1 participate in the formation of the address (4K instruction memory).

This type of branch does not engage the alu. On the condition that the jump control bit is 1, it concatenates the two literal values it receives from the command fields and passes the value to the program counter.

5.3 Branch, type 3

A third, trivial way to branch the program is to load system registers pch and pcl with literal values. The programmer first loads pch with the correct segment, which results in a branch to binary command:

segment:00000000, where "segment" is a 2-bit binary value.

A second command:

movi pcl, offset

placed at address (segment:00000000), loads pcl with the proper offset.

6. Subroutines

A eight-level-deep stack allows for subroutine calls. A "call <subroutine>" command makes an unconditional jump to the address in the subroutine field, while it pushes the PC+1 value into the stack.

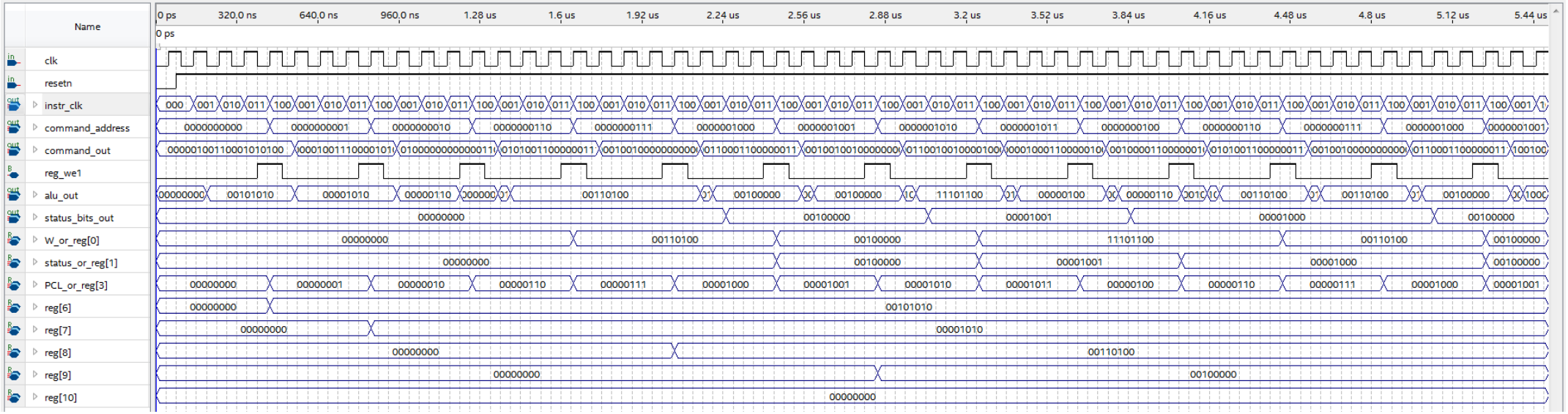
The subroutine field is the same absolute address field as in a jump command. In the case of a 10-bit program counter, it consists of the 2 LSBs in field 1 concatenated with the 8 bits in shamt & field 2 of the instruction word.

A return command (ret) pops the return address from the stack and loads it to the program counter. In this way, the program execution continues seamlessly.

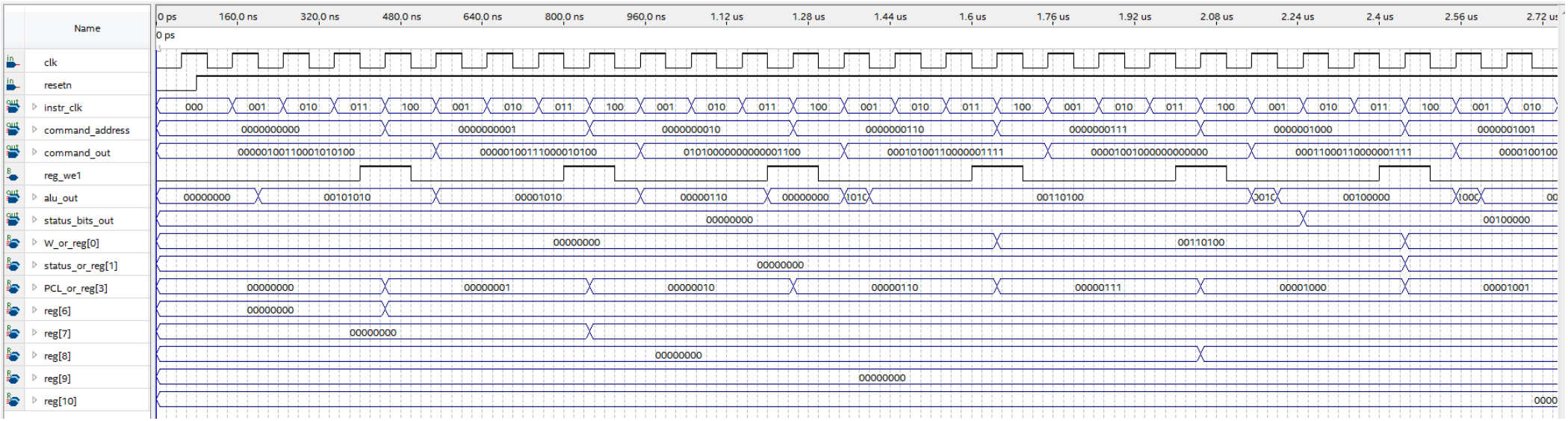
5.4 Instruction cycle and timing diagrams

functional simulation of several consecutive commands (unit robin_proc.vhd):

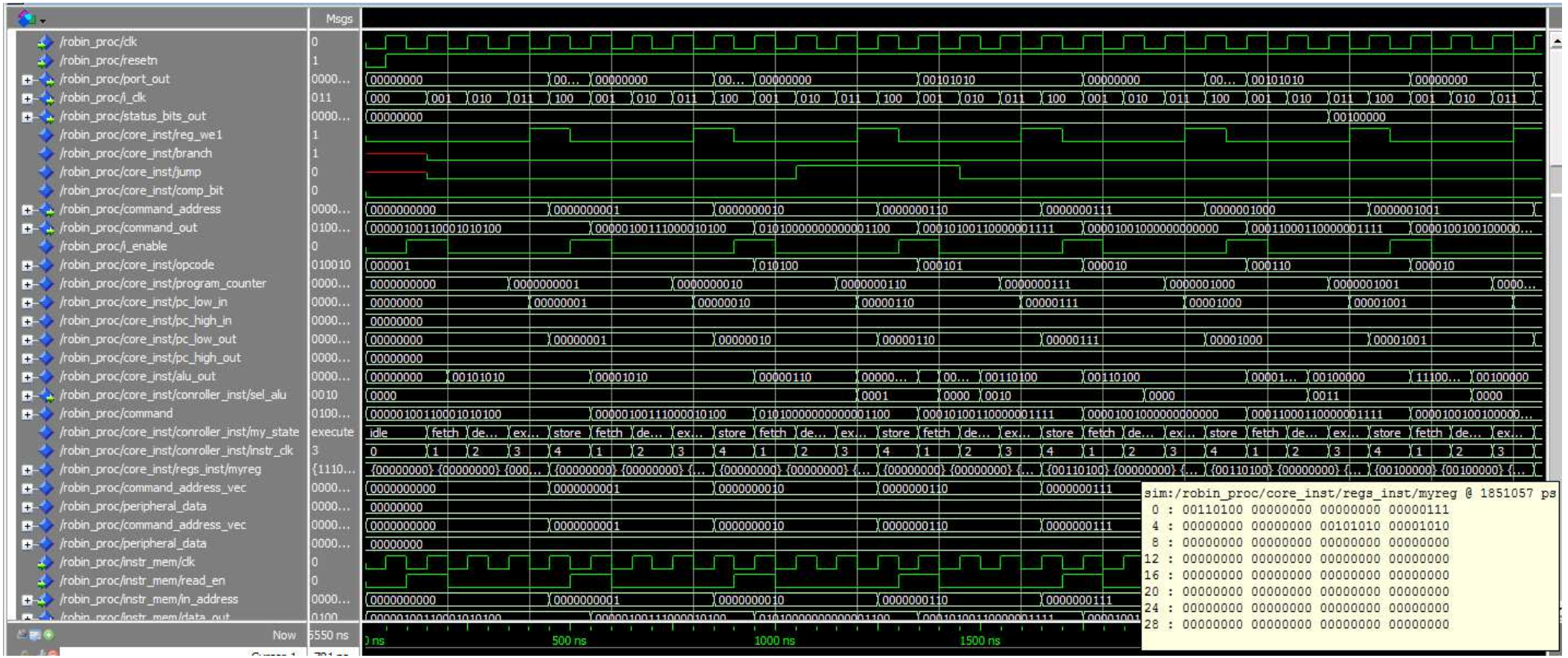
```
0 => "0000010011000010100",  --  reg1 <- literal     (6 <- 00101010) (52)
1 => "0000010011100010100",  --  reg1 <- literal     (7 <- 00001010) (10)
2 => "0101000000000001100",  --  jump to absolute address 00000110 (6)
3 => "0100010011000001101",  --  cmp reg1, reg2 ([6], [6]) fb=1
4 => "0100100001100000100",  --  branch 2 instructions below (or goto 6): pc <- pc + offset
5 => "0000010011000000000",  --  [6] <- 00000000
6 => "0001010011000001111",  --   W <- reg1 + reg2 (0 <- [6]+[7])
7 => "0000100100000000000",  --   reg1 <- reg2              (8 <- W) 62
8 => "0001100011000001111",  --   W <- reg1 - reg2 (0 <- [6]-[7])
9 => "0000100100100000000",  --   reg1 <- reg2              (9 <- W) 42: 00101010
10 => "0001100100100010001", --   W <- [9] - [8]
11 => "0000010001100001000", --   [3] <- 4 --load pcl with 4
12 => "0000100101000000000", --   (10 <- W) -20: 11101100
```



simulation zoom-in



Simulation of the same program using ModelSim. Register file at lower right is captured at the end of command 6.



7. Peripherals

APPENDIX A: VHDL Files for the main processor units

--Robinson Package: Declaration of components

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
PACKAGE robinson_package IS
--State enumeration and declaration
TYPE state IS (idle, fetch, decode, execute, store);
--COMPONENT DECLARATIONS
COMPONENT robin_fsm IS
GENERIC (N: NATURAL :=8);--Number of register bits
        PORT (  clk, resetn             : IN STD_LOGIC;
                opcode                  : IN STD_LOGIC_VECTOR(5 downto 0);
                 instr_mem_en           : OUT STD_LOGIC;
                 reg_we1, ctrl_lit, branch: OUT STD_LOGIC;
                 ctrl_busA              : OUT STD_LOGIC_VECTOR(1 downto 0);
                 sel_alu                : OUT STD_LOGIC_VECTOR(3 downto 0);
                 command_clk            : OUT STD_LOGIC_VECTOR(2 downto 0));
END COMPONENT;
COMPONENT    reg_file IS
GENERIC  (N: NATURAL:=8;        -- 8 bit data bus
          M: NATURAL :=32); -- Register file depth (M/4 are user regs, M/4 are system regs and M/2 are peripheral regs
        PORT (clk: IN std_logic;
                  reg_we1       : IN  std_logic;
                  rd_address1   : IN  std_logic_vector(4 downto 0);          -- total memory file is M regs
                  rd_address2   : IN  std_logic_vector(4 downto 0);
                  wr_address    : IN  std_logic_vector(4 downto 0);          --write address1
                  data_in       : IN  std_logic_vector(N-1 downto 0);        --Bus A
                  status        : IN  std_logic_vector(N-1 downto 0);
```

```
                pc_low          : IN  std_logic_vector(N-1 downto 0);
                int_flags       : IN  std_logic_vector(N-1 downto 0);
                data_out1       : OUT std_logic_vector(N-1 downto 0);  --Bus B
                data_out2       : OUT std_logic_vector(N-1 downto 0)); --Bus C
END COMPONENT;
COMPONENT decoder_unit IS
GENERIC(N: NATURAL :=8);--Number of register bits
        PORT (instr_word        : IN STD_LOGIC_VECTOR(19 downto 0);
                address1, address2, target_address: OUT STD_LOGIC_VECTOR(4 downto 0);
                 opcode         : OUT STD_LOGIC_VECTOR(5 downto 0);
                fun_bit         : OUT STD_LOGIC; --function bit
                shift           : OUT STD_LOGIC_VECTOR(2 downto 0);
                literval        : OUT STD_LOGIC_VECTOR(N-1 downto 0));--literal value
END COMPONENT;
COMPONENT alu_unit is
        generic(N: NATURAL:=8); --number of bits
        port(resetn : IN STD_LOGIC;
                busD, busB :IN std_logic_vector(N-1 downto 0);                                  --input signals
                sel_alu: IN STD_LOGIC_VECTOR(3 downto 0);                                       --selection line alu
                shift: IN STD_LOGIC_VECTOR(2 downto 0);                                         --shift amount
                cin: IN STD_LOGIC; --carry in
                instr_clk: IN STD_LOGIC_VECTOR(2 downto 0);                                     --instruction clock
                alu_out: OUT std_logic_vector(N-1 downto 0);                                    --results
                status_bits: OUT STD_LOGIC_VECTOR(N-1 downto 0) :="00000000");    --status_reg bits (gt, eq, lt, zero, carry)
END COMPONENT;
COMPONENT  instruction_memory IS
GENERIC(in_depth: NATURAL := 10);--Normal depth of instruction memory is 2^10=1024
        PORT( clk, read_en: IN STD_LOGIC;
                        in_address: IN STD_LOGIC_VECTOR(in_depth-1 downto 0);--INTEGER RANGE 0 TO in_depth-1;
                        data_out: OUT STD_LOGIC_VECTOR(19 downto 0));--20 bits instructions
```

```
END component;
END PACKAGE;
```

Instruction decode unit

```
--This unit receives as input the instruction word and produces the various fields (opcode, reg addresses,
--parameteres, literals) which are necessary for the operation of the register file and the control fsm.
----------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
USE work.robinson_package.all;
----------------------------
ENTITY decoder_unit IS
GENERIC(N: NATURAL :=8);                                                    --Number of register bits
        PORT (instr_word        : IN STD_LOGIC_VECTOR(19 downto 0);
                address1, address2, target_address: OUT STD_LOGIC_VECTOR(4 downto 0);       -- register in/out addresses
                opcode          : OUT STD_LOGIC_VECTOR(5 downto 0);
                fun_bit         : OUT STD_LOGIC;                            --function bit
                shift           : OUT STD_LOGIC_VECTOR(2 downto 0);
                literval        : OUT STD_LOGIC_VECTOR(N-1 downto 0));      --literal value
END decoder_unit;

ARCHITECTURE decode OF decoder_unit IS

SIGNAL fun_bit1         : STD_LOGIC;
SIGNAL reg1, reg2  : STD_LOGIC_VECTOR(4 downto 0);
BEGIN
--instruction decode signals--
opcode <= instr_word(19 downto 14);
```

```
fun_bit1 <= instr_word(0);
reg1 <= instr_word(13 downto 9);
reg2 <= instr_word(5 downto 1);
address1 <= reg1;
address2 <= reg2;
shift <= instr_word(8 downto 6);
literval <= instr_word(8 downto 1);

WITH fun_bit1 SELECT
        target_address <= reg1 WHEN '0',                    --select reg1 as target with fb=0
                          "00000" WHEN OTHERS;              --select W as target with fb=1
fun_bit <= fun_bit1;
END decode;
```

```
--REGISTER FILE for Robin processor
--Robotics and Intelligent Systems Lab
--Date: 25-4-2019
---------------------------------------
--Multi port memory file with 32 registers
--16 general registes
--6 system registers
--10 peripheral registers
---------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY reg_file IS
GENERIC  (N: NATURAL:=8;   --8 bit data bus
            M: NATURAL :=32);--Register file depth
          PORT (clk: IN std_logic;
                    reg_we1                  : IN  std_logic;
                    rd_address1     : IN  std_logic_vector(4 downto 0);
                    rd_address2     : IN  std_logic_vector(4 downto 0);
                    wr_address      : IN  std_logic_vector(4 downto 0);
                    data_in         : IN  std_logic_vector(N-1 downto 0);  --Bus A
                    status          : IN  std_logic_vector(N-1 downto 0);  --system reg
                    pc_low          : IN  std_logic_vector(N-1 downto 0);  --system reg
                    int_flags       : IN  std_logic_vector(N-1 downto 0);  --system reg
                    data_out1       : OUT std_logic_vector(N-1 downto 0);  --Bus B
                    data_out2       : OUT std_logic_vector(N-1 downto 0)); --Bus C
END reg_file;
```

```
ARCHITECTURE reg OF reg_file IS

TYPE registers IS ARRAY (0 TO M-1) OF std_logic_vector(N-1 downto 0);
SIGNAL myreg: registers :=(OTHERS=>"00000000");
SIGNAL wr_addr: integer range 0 to M-1;
SIGNAL rd_addr1, rd_addr2: integer range 0 to M-1;

BEGIN

wr_addr  <= to_integer(unsigned(wr_address));
rd_addr1 <= to_integer(unsigned(rd_address1));
rd_addr2 <= to_integer(unsigned(rd_address2));

        PROCESS(clk)
        BEGIN
                IF (clk'event AND clk = '1') THEN --store with positive edge
                        IF (reg_we1='1') THEN
                                myreg(wr_addr) <= data_in; --destination reg: reg1 or W
                                myreg(1) <= status;
                                myreg(2) <= int_flags;
                                myreg(3) <= pc_low;
                        END IF;
                END IF;
        END PROCESS;
data_out1 <= myreg(rd_addr1);
data_out2 <= myreg(rd_addr2);
END reg;
```

```
--Robin Processor ALU
--Robotics and Intelligent Systems Lab
--Version 0, date 5/8/2019
--Arithmetic and logic unit for robin processor
------------------------------
Library ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
------------------------------
entity alu_unit is
	generic(N: NATURAL:=8);                                                    --number of register bits
	port(resetn : IN STD_LOGIC;
		busD, busB :IN std_logic_vector(N-1 downto 0);        --input signals
		sel_alu: IN STD_LOGIC_VECTOR(3 downto 0);             --selection line alu
		shift: IN STD_LOGIC_VECTOR(2 downto 0);               --shift amount
		cin: IN STD_LOGIC;                                                     --carry in
		instr_clk: IN STD_LOGIC_VECTOR(2 downto 0);           --NATURAL RANGE 0 TO 5; --instruction clock
		alu_out: OUT std_logic_vector(N-1 downto 0);          --results
		status_bits: OUT STD_LOGIC_VECTOR(N-1 downto 0));--status_reg bits (gt, eq, lt, zero, carry)
end entity;
------------------------------
architecture robin_alu OF alu_unit IS
signal alu_res: STD_LOGIC_VECTOR(N downto 0);
signal alu_out1, status_bits1: STD_LOGIC_VECTOR(N-1 downto 0):="00000000";
signal zero, carry, ov_add, ov_sub, gt, eq, lt, ls: STD_LOGIC;
SIGNAL shift_nat: NATURAL RANGE 0 TO N-1;
BEGIN

shift_nat <= to_integer(unsigned(shift));
```

```
WITH sel_alu SELECT
	alu_res <= std_logic_vector(unsigned('0' & busD)) WHEN "0000", --busD passthrough
		std_logic_vector(unsigned('0' & busB)) WHEN "0001", --busB passthrough
		std_logic_vector(unsigned('0' & busB) + unsigned('0' & busD)) WHEN "0010", -- add busB, busD
		std_logic_vector(unsigned('0' & busB) - unsigned('0' & busD)) WHEN "0011", -- sub busB, busD (unsigned subtraction) and compare (cmp)
		std_logic_vector(unsigned('0' & busB) + unsigned('0' & busD) + ('0' & cin)) WHEN "0100", -- add busB, busD with carry
		std_logic_vector(unsigned('0' & busB) - unsigned('0' & busD) - ('0' & cin)) WHEN "0101", -- sub busB, busD with borrow
		'0' & (busB AND busD) WHEN "0110",
		'0' & (busB OR busD) WHEN "0111",
		'0' & to_stdlogicvector(to_bitvector(busD) SLL shift_nat) WHEN "1000", --shift left logic
		'0' & to_stdlogicvector(to_bitvector(busD) SRL shift_nat) WHEN "1001", --shift right logic
		"000000000" WHEN OTHERS;

alu_out1 <= alu_res(N-1 downto 0);
carry <= alu_res(N); --carry flag for unsigned operations
zero <= '1' when alu_out1 = "00000000" else '0';
gt <= carry XNOR zero; eq <= zero; lt <= carry; --comparison bits
ls <= busD(7); --literal/offset sign bit
ov_add <= (busB(N-1) AND busD(N-1) AND (NOT(alu_res(N-1)))) OR (NOT(busB(N-1)) AND NOT(busD(N-1)) AND alu_res(N-1)); --compute overflow for signed addition
ov_sub <= (NOT(busB(N-1)) AND busD(N-1) AND (alu_res(N-1))) OR (busB(N-1) AND NOT(busD(N-1)) AND NOT(alu_res(N-1))); --compute overflow for signed subtraction
alu_out <= alu_out1;

--Configure status bits
PROCESS(instr_clk, sel_alu, resetn)
BEGIN
	IF resetn = '0' then
			status_bits1 <= (OTHERS => '0'); --initiate statusbits to zero
	ELSIF (instr_clk = "011") THEN
		IF (sel_alu = "0010") THEN
			status_bits1 <= ls & status_bits1(6 downto 3) & ov_add & zero & carry; --carry;   -- status_bits reg in ALU
```

```
                ELSIF (sel_alu = "0011") THEN
                        status_bits1 <= ls & status_bits1(6) & gt & eq & lt & ov_sub & zero & carry; --carry;
                ELSIF (sel_alu = "1000") THEN
                        status_bits1 <= '0' & busD(N-1) & status_bits1(5 downto 0); --shift left bit in status
                ELSIF (sel_alu = "1001") THEN
                        status_bits1 <= '0' & busD(0) & status_bits1(5 downto 0); --shift right bit in status
        END IF;
END IF;
END PROCESS;
status_bits <= status_bits1;
end robin_alu;
```

```
--Main control unit for Robin processor
--By Robotics and Intelligent Systems Lab
--Date: 26-4-2019
--update: 16-8-2019
--main Robin controller fsm
--fsm and control

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
--library xil_defaultlib;
USE work.robinson_package.all;
----------------------------
ENTITY robin_fsm IS
GENERIC (N: NATURAL :=8);--Number of register bits
        PORT (clk, resetn           : IN STD_LOGIC;
                        opcode : IN STD_LOGIC_VECTOR(5 downto 0);
                instr_mem_en            : OUT STD_LOGIC;
                reg_we1, ctrl_lit, branch: OUT STD_LOGIC;
                        ctrl_busA : OUT STD_LOGIC_VECTOR(1 downto 0);
                sel_alu : OUT STD_LOGIC_VECTOR(3 downto 0);
                        command_clk: OUT STD_LOGIC_VECTOR(2 downto 0));--NATURAL RANGE 0 TO 5 :=0;
END robin_fsm;

ARCHITECTURE controller OF robin_fsm IS
--Declaration of internal Control signals
SIGNAL new_reg_we1: STD_LOGIC;
SIGNAL new_ctrl_busA: STD_LOGIC_VECTOR(1 downto 0);--selection line mux busA
SIGNAL new_sel_alu: STD_LOGIC_VECTOR(3 downto 0);--selection line alu
SIGNAL new_ctrl_lit, new_instr_mem_en: STD_LOGIC; --sel line mux busC/literal;
```

```
--State Signals
SIGNAL pr_state, nx_state, my_state: state;
--System signals
SIGNAL instr_clk: NATURAL RANGE 0 TO 5 :=0;
BEGIN
---Sequential Section of FSM---
	PROCESS(clk, resetn)
	BEGIN
		IF (resetn = '0') THEN
		  pr_state <= idle;
		  instr_clk <= 0;
		ELSIF (clk'EVENT AND clk = '1') THEN
		  instr_clk <= instr_clk + 1; --increment instruction clock
		  pr_state <= nx_state;
		  IF (instr_clk = 4) THEN
		    instr_clk <= 1;
		  END IF;
		END IF;
	END PROCESS;
--Combinational Section of FSM---
	PROCESS (pr_state)
	BEGIN
		CASE pr_state IS

			WHEN idle => --do nothing/initialize control signals
				new_instr_mem_en <= '1';
				new_reg_we1 <= '0';  --reg and memory enable bits
				new_ctrl_busA <= "00"; new_sel_alu <= "0000"; new_ctrl_lit <= '0'; --mux selection bits

				IF (instr_clk = 0) THEN
```

```
                nx_state <= fetch;
        ELSE
                nx_state <= idle;
        END IF;
WHEN fetch =>                   --present new instruction word
        new_instr_mem_en <= '0';
        new_reg_we1<='0';

        CASE opcode IS
                WHEN "000001" => --move reg1<-literal (movi reg1, literal)
                        new_ctrl_busA <= "10"; new_sel_alu<="0000";new_ctrl_lit <='1'; branch <= '0';

                WHEN "000010" => --move reg1<-reg2 (mov reg1, reg2)
                        new_ctrl_busA <= "10"; new_sel_alu<="0000";new_ctrl_lit <='0'; branch <= '0';

                WHEN "000101" => --add reg1, reg2, x (reg1 <- reg1 + reg2, x=1)
                        new_ctrl_busA <= "10"; new_sel_alu<="0010";new_ctrl_lit <='0'; branch <= '0';

                WHEN "000110" => --sub reg1, reg2 (reg1 <- reg1-reg2)
                        new_ctrl_busA <= "10"; new_sel_alu<="0011";new_ctrl_lit <='0'; branch <= '0';

                WHEN "000111" => --addc reg1, reg2, fb (reg1 <- reg1 + reg2 + cin)
                        new_ctrl_busA <= "10"; new_sel_alu<="0100";new_ctrl_lit <='0'; branch <= '0';

                WHEN "001000" => --subc reg1, reg2 (reg1 <- reg1-reg2 - cin)
                        new_ctrl_busA <= "10"; new_sel_alu<="0101";new_ctrl_lit <='0'; branch <= '0';

                WHEN "001011" => --and reg1, reg2 (reg1 <- reg1 AND reg2)
                        new_ctrl_busA <= "10"; new_sel_alu<="0110";new_ctrl_lit <='0'; branch <= '0';
```

```
                WHEN "001100" => --or reg1, reg2 (reg1 <- reg1 OR reg2)
                        new_ctrl_busA <= "10"; new_sel_alu<="0111";new_ctrl_lit <='0'; branch <= '0';

                WHEN "001111" => --shift left reg1 (reg1 << shift)
                        new_ctrl_busA <= "10"; new_sel_alu<="1000";new_ctrl_lit <='0'; branch <= '0';

                WHEN "010000" => --shift right reg1 (reg1 >> shift)
                        new_ctrl_busA <= "10"; new_sel_alu<="1001";new_ctrl_lit <='0'; branch <= '0';

                WHEN "010001" => --cmp reg1, reg2 (set bits gt/eq/lt)
                        new_ctrl_busA <= "10"; new_sel_alu<="0011";new_ctrl_lit <='0'; branch <= '0';

                WHEN "010010" => -- branch on equal (after a cmp). It adds literal addr to PC reg (relocatable, <256 commands)
                        new_ctrl_busA <= "10"; new_sel_alu<="0010";new_ctrl_lit <='1'; branch <= '1';

                WHEN OTHERS => --this is the end command
                        new_ctrl_busA <= "11"; new_sel_alu<="0000";new_ctrl_lit<='0'; branch <= '0';--pass zero

        END CASE;
        IF (instr_clk = 1) THEN
                nx_state <= decode;
        ELSE
                nx_state <= fetch;
        END IF;

WHEN decode => --make control bits ready
new_instr_mem_en <= '0';
new_reg_we1<='0';

        CASE opcode IS
```

```
WHEN "000001" => --move reg1<-literal
        new_ctrl_busA <= "10"; new_sel_alu<="0000";new_ctrl_lit <= '1'; branch <= '0';--enable literal and pass it through

WHEN "000010" => --move reg1<-reg2 (mov reg1, reg2)
        new_ctrl_busA <= "10"; new_sel_alu<="0000"; new_ctrl_lit <= '0'; branch <= '0'; --passthrough busC


WHEN "000101" => --add reg1, reg2, x (reg1 <- reg1 + reg2, x=0)
        new_ctrl_busA <= "10"; new_sel_alu<="0010";new_ctrl_lit<='0'; branch <= '0';

WHEN "000110" => --sub reg1, reg2 (reg1 <- reg1-reg2)
        new_ctrl_busA <= "10"; new_sel_alu<="0011";new_ctrl_lit<='0'; branch <= '0';

WHEN "000111" => --addc reg1, reg2, fb (reg1 <- reg1 + reg2 + cin)
        new_ctrl_busA <= "10"; new_sel_alu<="0100";new_ctrl_lit<='0'; branch <= '0';

WHEN "001000" => --subc reg1, reg2 (reg1 <- reg1-reg2 - cin)
        new_ctrl_busA <= "10"; new_sel_alu<="0101";new_ctrl_lit<='0'; branch <= '0';

WHEN "001011" => --and reg1, reg2 (reg1 <- reg1 AND reg2)
        new_ctrl_busA <= "10"; new_sel_alu<="0110";new_ctrl_lit<='0'; branch <= '0';

WHEN "001100" => --or reg1, reg2 (reg1 <- reg1 OR reg2)
        new_ctrl_busA <= "10"; new_sel_alu<="0111";new_ctrl_lit<='0'; branch <= '0';

WHEN "001111" => --shift left reg1 (reg1 << shift)
        new_ctrl_busA <= "10"; new_sel_alu<="1000";new_ctrl_lit<='0'; branch <= '0';

WHEN "010000" => --shift right reg1 (reg1 >> shift)
        new_ctrl_busA <= "10"; new_sel_alu<="1001";new_ctrl_lit<='0'; branch <= '0';
```

```
                WHEN "010001" => --cmp reg1, reg2 (set bits gt/eq/lt)
                        new_ctrl_busA <= "10"; new_sel_alu<="0011";new_ctrl_lit<='0'; branch <= '0';

                WHEN "010010" => -- branch on equal (after a cmp). It adds literal offset to PC reg (low)
                        new_ctrl_busA <= "10"; new_sel_alu<="0010";new_ctrl_lit<='1'; branch <= '1';

                WHEN OTHERS =>
                        new_ctrl_busA <= "11"; new_sel_alu<="0000";new_ctrl_lit<='0'; branch <= '0';

        END CASE;
        IF (instr_clk = 2 AND opcode /= "111111") THEN
                nx_state <= execute;
        ELSE
                nx_state <= idle;
        END IF;

WHEN execute => --apply control bits for execution
new_instr_mem_en <= '0';
new_reg_we1<='1'; --write to register file

        CASE opcode IS
                WHEN "000001" => --reg1<-literal
                        new_ctrl_busA <= "10"; new_sel_alu<="0000"; new_ctrl_lit <= '1'; branch <= '0';

                WHEN "000010" => --move reg1<-reg2 (mov reg1, reg2)
                        new_ctrl_busA <= "10"; new_sel_alu<="0000"; new_ctrl_lit <= '0'; branch <= '0';

                WHEN "000101" => --add reg1, reg2, x (reg1 <- reg1 + reg2, x=1)
                        new_ctrl_busA <= "10"; new_sel_alu<="0010"; new_ctrl_lit <= '0'; branch <= '0';
```

```
WHEN "000110" => --sub reg1, reg2 (reg1 <- reg1-reg2)
        new_ctrl_busA <= "10"; new_sel_alu<="0011"; new_ctrl_lit <= '0'; branch <= '0';

WHEN "000111" => --addc reg1, reg2, fb (reg1 <- reg1 + reg2 + cin)
        new_ctrl_busA <= "10"; new_sel_alu<="0100"; new_ctrl_lit <= '0'; branch <= '0';

WHEN "001000" => --subc reg1, reg2 (reg1 <- reg1-reg2 - cin)
        new_ctrl_busA <= "10"; new_sel_alu<="0101"; new_ctrl_lit <= '0'; branch <= '0';

WHEN "001011" => --and reg1, reg2 (reg1 <- reg1 AND reg2)
        new_ctrl_busA <= "10"; new_sel_alu<="0110"; new_ctrl_lit <= '0'; branch <= '0';

WHEN "001100" => --or reg1, reg2 (reg1 <- reg1 OR reg2)
        new_ctrl_busA <= "10"; new_sel_alu<="0111"; new_ctrl_lit <= '0'; branch <= '0';

WHEN "001111" => --shift left reg1 (reg1 << shift)
        new_ctrl_busA <= "10"; new_sel_alu<="1000"; new_ctrl_lit <= '0'; branch <= '0';

WHEN "010000" => --shift right reg1 (reg1 >> shift)
        new_ctrl_busA <= "10"; new_sel_alu<="1001"; new_ctrl_lit <= '0'; branch <= '0';

WHEN "010001" => --cmp reg1, reg2 (set bits gt/eq/lt)
        new_ctrl_busA <= "10"; new_sel_alu<="0011"; new_ctrl_lit <= '0'; branch <= '0';

WHEN "010010" => -- branch on equal (after a cmp). It adds literal addr to PC reg
        new_ctrl_busA <= "10"; new_sel_alu<="0010"; new_ctrl_lit <= '1'; branch <= '1';

WHEN OTHERS =>
        new_ctrl_busA <= "11"; new_sel_alu<="0000"; new_ctrl_lit <= '0'; branch <= '0';
```

```
		END CASE;
		IF (instr_clk = 3) THEN
			nx_state <= store;
		ELSE
			nx_state <= execute;
		END IF;

	WHEN store => --store results
	new_instr_mem_en <= '1'; -- make next instruction ready
	new_reg_we1<='1';

		CASE opcode IS
			WHEN "000001" => --reg1<-literal
				new_ctrl_busA <= "10"; new_sel_alu <= "0000"; new_ctrl_lit <= '1'; branch <= '0';

			WHEN "000010" => --move reg1<-reg2 (mov reg1, reg2)
				new_ctrl_busA <= "10"; new_sel_alu <= "0000"; new_ctrl_lit <= '0'; branch <= '0';

			WHEN "000101" => --add reg1, reg2, x (reg1 <- reg1 + reg2, x=1)
				new_ctrl_busA <= "10"; new_sel_alu <= "0010"; new_ctrl_lit <= '0'; branch <= '0';

			WHEN "000110" => --sub reg1, reg2 (reg1 <- reg1-reg2)
				new_ctrl_busA <= "10"; new_sel_alu <= "0011"; new_ctrl_lit <= '0'; branch <= '0';

			WHEN "000111" => --addc reg1, reg2, fb (reg1 <- reg1 + reg2 + cin)
				new_ctrl_busA <= "10"; new_sel_alu <= "0100"; new_ctrl_lit <= '0'; branch <= '0';

			WHEN "001000" => --subc reg1, reg2 (reg1 <- reg1-reg2 - cin)
				new_ctrl_busA <= "10"; new_sel_alu <= "0101"; new_ctrl_lit <= '0'; branch <= '0';
```

```
                WHEN "001011" => --and reg1, reg2 (reg1 <- reg1 AND reg2)
                        new_ctrl_busA <= "10"; new_sel_alu <= "0110"; new_ctrl_lit <= '0'; branch <= '0';

                WHEN "001100" => --or reg1, reg2 (reg1 <- reg1 OR reg2)
                        new_ctrl_busA <= "10"; new_sel_alu <= "0111"; new_ctrl_lit <= '0'; branch <= '0';

                WHEN "001111" => --shift left reg1 (reg1 << shift)
                        new_ctrl_busA <= "10"; new_sel_alu <= "1000"; new_ctrl_lit <= '0'; branch <= '0';

                WHEN "010000" => --shift right reg1 (reg1 >> shift)
                        new_ctrl_busA <= "10"; new_sel_alu <= "1001"; new_ctrl_lit <= '0'; branch <= '0';

                WHEN "010001" => --cmp reg1, reg2 (set bits gt/eq/lt)
                        new_ctrl_busA <= "10"; new_sel_alu <= "0011"; new_ctrl_lit <= '0'; branch <= '0';

                 WHEN "010010" => -- branch on equal (after a cmp). It adds literal addr to PC reg. fb=0
                        new_ctrl_busA <= "10"; new_sel_alu <= "0010"; new_ctrl_lit <= '1'; branch <= '1';

                WHEN OTHERS =>
                        new_ctrl_busA <= "11"; new_sel_alu <= "0000"; new_ctrl_lit <= '0'; branch <= '0';

        END CASE;

        IF (instr_clk = 4) THEN
                nx_state <= fetch;
        ELSE
                nx_state <= store;
        END IF;
    END CASE;
END PROCESS;
```

```
--Output Section--
PROCESS (clk, resetn) --use negative edge to synchronize control bits
BEGIN
		IF(resetn = '0') THEN
			reg_we1 <= '0';
			ctrl_busA <= "00"; sel_alu<="0000";ctrl_lit<='0';
			instr_mem_en <= '0';
		ELSIF(clk'EVENT AND clk = '0') THEN
			reg_we1 <= new_reg_we1; instr_mem_en <= new_instr_mem_en;
			ctrl_busA <= new_ctrl_busA; sel_alu <= new_sel_alu; ctrl_lit <= new_ctrl_lit;
		END IF;
END PROCESS;
--final assignments
command_clk <= std_logic_vector(to_unsigned(instr_clk, 3));
my_state <= pr_state;
END controller;
```

```
--This is a top level entity for the instantiation of robin processor
---------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.robinson_package.all;
---------------------------------
entity robin_proc is
generic(N: natural :=8);
        port(port_in: in std_logic_vector(N-1 downto 0);
            clk, resetn: in std_logic;
            port_out: out std_logic_vector(N-1 downto 0);
                i_clk: OUT STD_LOGIC_VECTOR(2 downto 0);
                comp_bit_out, branch_out: OUT STD_LOGIC;                    --to be removed in the end
                status_bits_out: OUT STD_LOGIC_VECTOR(N-1 downto 0); --to be removed in the end
                alu_hello: OUT STD_LOGIC_VECTOR(N-1 downto 0);          --to be removed in the end
                command_address: OUT std_logic_vector(9 downto 0); --to be removed in the end
                command_out: OUT std_logic_vector(19 downto 0));            --to be removed in the end
end robin_proc;
---------------------------------
architecture inst of robin_proc is

signal command: std_logic_vector(19 downto 0);
signal i_enable: std_logic;
--signal command_address_int: integer range 0 to 1024; --std_logic_vector(9 downto 0);
signal command_address_vec: std_logic_vector(9 downto 0);
signal peripheral_data: STD_LOGIC_VECTOR(N-1 downto 0) := "00000000";
--signal my_result: std_logic_vector(7 downto 0);
```

```
COMPONENT robin_core IS
GENERIC(N: NATURAL :=8);--Number of register bits
        PORT (clk, resetn               : IN STD_LOGIC;
                instr_word              : IN STD_LOGIC_VECTOR(19 downto 0);  --from program memory
                peripheral_bus          : IN STD_LOGIC_VECTOR(N-1 downto 0); --peripheral data
                instr_mem_en            : OUT STD_LOGIC;                                        -- read program memory control bit
                i_address                       : OUT STD_LOGIC_VECTOR(9 downto 0);  --instruction address
                result                          : OUT STD_LOGIC_VECTOR(N-1 downto 0);--busB output from reg file, to be removed in the end
                i_clk                           : OUT STD_LOGIC_VECTOR(2 downto 0);  --only for viewing, to be removed in the end
                comp_bit_out, branch_out: OUT STD_LOGIC;                                --to be removed in the end
                status_bits_out: OUT STD_LOGIC_VECTOR(N-1 downto 0); --to be removed in the end
                alu_hello: OUT STD_LOGIC_VECTOR(N-1 downto 0));             --to be removed in the end
END COMPONENT;
---------------------------------
begin
command_address <= command_address_vec;
command_out <= command;
--
instr_mem: instruction_memory
generic map(in_depth => 10)
port map(clk => clk, read_en =>i_enable, in_address => command_address_vec, data_out => command);

core_inst: robin_core
generic map(N => 8)
port map(clk => clk, resetn => resetn, instr_word => command, peripheral_bus => peripheral_data, instr_mem_en => i_enable,
i_address => command_address_vec, result => port_out, i_clk => i_clk, comp_bit_out => comp_bit_out, branch_out => branch_out,
status_bits_out => status_bits_out, alu_hello => alu_hello);

end inst;
```